02614 HIGH-PERFORMANCE COMPUTING

Assignment 1: Matrix Multiplication

Authors:

Jules Belveze s182291

Marina Pons Mojena s182288

Daniel Brian Thoren s144222

Alexandros Dorodoulis s182148

January 11, 2019

Contents

1 Introduction

Through this assignment we develop a library of functions that carry out matrix-matrix multiplication. When multiplying a matrix, A of size $m \times k$ with matrix, B of size $k \times n$, we obtain a matrix, C of size $m \times n$. Multiplying matrices is described by:

$$C_{ij} = \sum_{p=1}^{k} A_{ip} B_{pj} \tag{1}$$

The performance is studied and optimized with several different techniques:

- Permutations of the loop order
- The blocking method
- Compiler optimization

This is all done in `C` which has a row-major format (contrary to *Fortran*). Therefore we are using double-array-indexing for the sake of simplicity and readability. Using C we know that the matrices are stored and accessed row-wise. This is essential as it determines how elements are fetched from memory and how the cache-use can be optimized.

2 Native and library (CBLAS) matrix multiplication

First matrix multiplication is implemented using the function: `mutmalt_nat` which carries out the usual algebraic matrix multiplication: Computing the sum elementwise column-row multiplication:

```
void matmult_nat(int m, int n, int k, double **A, double **B, double **C){
    int i, j, p;
    for (i = 0; i < m; i++){
        for (j = 0; j < n; j++){
            C[i][j] = 0;
            for (p = 0; p < k; p++){
                C[i][j] += A[i][p] * B[p][j];
            }
        }
    }
}
```

The correctness is verified using a hash return zero assertion. Thereafter the `dgemm()` subroutine from the *CBLAS* library is implemented:

```
void matmult_lib(int m, int n, int k, double **A, double **B, double **C){
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, k, 1.0, A[0], k, B[0], n, 0.0, C[0], n);
}
```

The matrices A,B and C are initialized as `A[0],B[0] and C[0]` because the subroutine `dgemm()` takes one dimensional arrays as an input, instead of two dimensional.

The implementation is again verified with the third output equal to 0. The performance of `dgemm()` is superior to the native implementation.

Assigning the following values: $m = n = k = 500$, yields a native implementation performance of 1673.8 MFlop/s whereas the library function performance at: 13837.6 MFlop/s - which is significantly bigger.

3 Permutation

As observed in the `multmat_nat` function, three nested `for` loops compute the matrix multiplication. By changing the order of the loops (and making the appropriate index changes), performance can be optimized. In the six different permutations obtained, the performance varies due to the order in which access the data - by column or by rows.

The functions are called `multmat_mnk`, `multmat_mkn`... such that the first letter indicates the outer-most loop, and the third one the inner-most loop. The code for `multmat_mnk` is shown and is analogous to the other permutations.

```
void matmult_mnk(int m, int n, int k, double **A, double **B, double **C){
    double start, cpu_time_used;
    start = (double) clock();
    int i, j, p;

    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
                C[i][j] = 0.0;

    for (i = 0; i < m; i++){
        for (j = 0; j < n; j++){
            for (p = 0; p < k; p++){
                C[i][j] += A[i][p] * B[p][j];
}   }   }   }
```

It is important to highlight that if the k-loop is the inner-most loop, we can not initialize the C matrix inside the loop as in `matmult_nat`. Instead the initialization of the C matrix is done externally so that the six different permutation functions are compared fairly. Again, all functionas are verified for correctness.

Matrix multiplication is performed for different matrix with each permutations of loop orders in the `hpcintro` to get comparable results. The aim is to calculate the memory used by our functions. To make it simple, square matrices are used to perform calculations. In order to calculate memory consumption, we use that a `double` takes 8 bytes, thus the total memory is: $3 \cdot 8 \cdot m^2 = 24 \cdot m^2$. If we check the sizes for the L1, L2 and L3 caches we get that they are: 32KB, 1024KB and 19712KB respectively. That is, we are interested in matrix sizes around: $m = 36$, $m = 206$ and $m = 906$. If we use these sizes we will access the three different levels of the cache, meaning that we will be able to study the performance. A part from these three values we also take values between them.

First of all we do not apply any optimization to see the raw comparison between the different permutations.

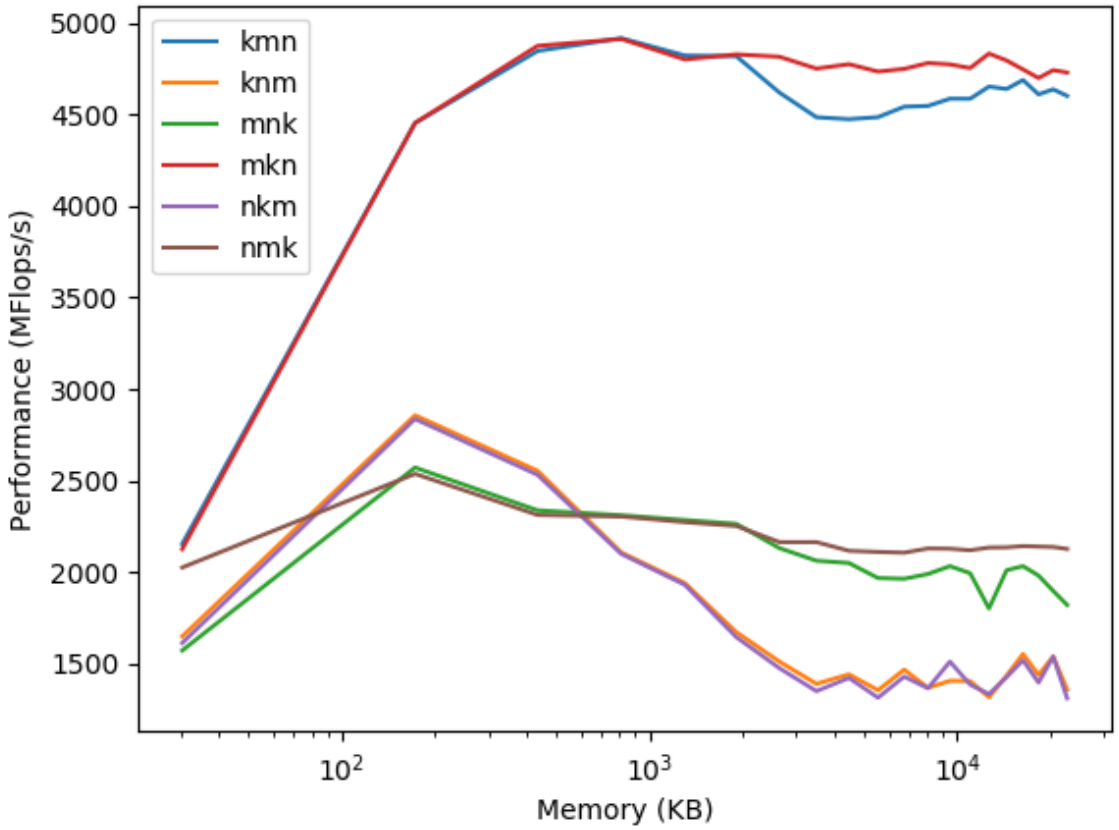

Figure 1: Performance for the different permutations without the optimizations from the compiler.

We can observe in Fig.1 that there are small peaks as we pass through the different cache levels. Thus, the best nested orders are less affected by the matrix size (memory footprint) while the other ones show a huge decrease regarding performance as the matrix size increases.

Rerunning the permutations with the compiler optimization produces the following results.

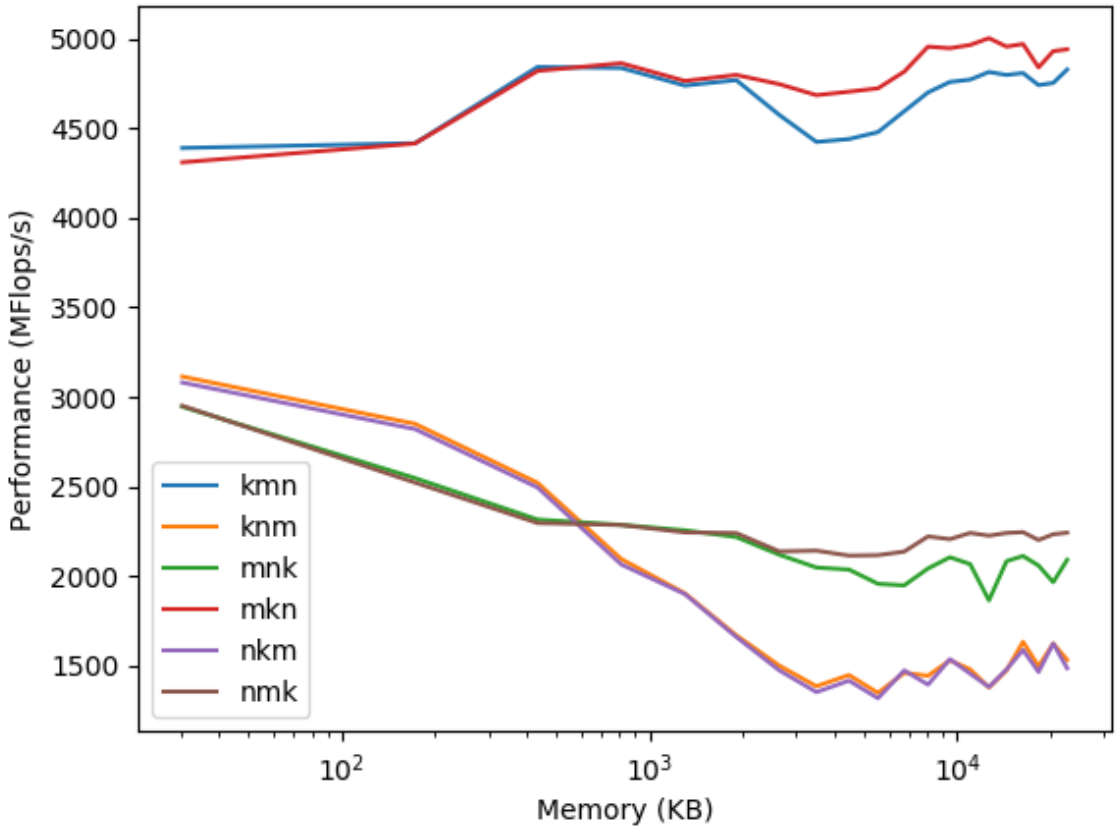

Figure 2: Performance for the different permutations using the optimizations from the compiler.

From Fig.2, it's clear that there is a huge difference in the performance of the program at the beginning of the graph and afterwards the performance is similar to Fig.1.

3.1 Loop order

In C, matrices are stored row-wise in the memory, and also fetched row-wise. This means that every time we are seeking for a matrix value and it is not in the cache we then save a full cache line of values in a row-wise way. Thus, the preferable order of the loop is the one that goes through the rows in the inner loop. Hence, the information that we are looking for is already in the cache, therefore we avoid cache misses. This justifies why the `kmn` and the `mkn` are the loops that give a better performance. In the `mkn` case we using the values saved on the cache as we are going through `n` in the inner loop (B rows and C rows). We can observe in Fig.1 that the best loop orders are the ones that have `n` as the most inner loop.

Using time as a metric, the increase in runtime can be plotted against matrix size. Fig.3 shows permutation variations for the `Ofast` optimization Fig.4 shows the best permutation variations for each mode of optimization. We can see that `Ofast + mkn` has the best performance.

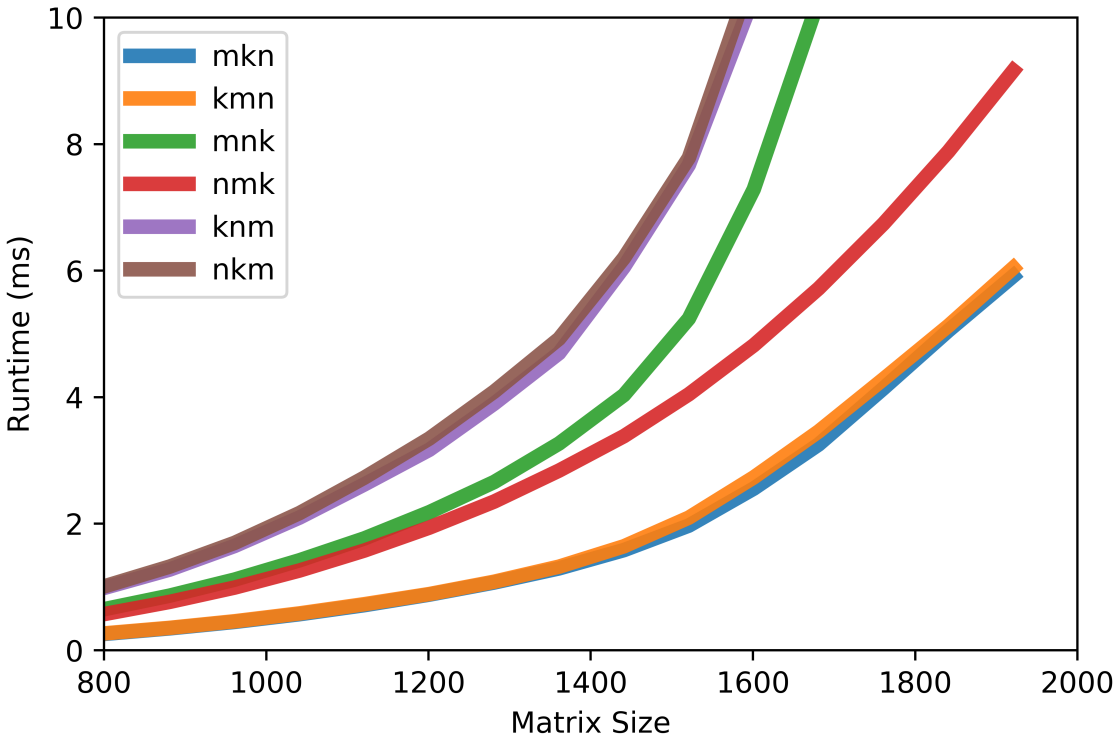

Figure 3: Runtime for the different permutations using `Ofast` optimization.

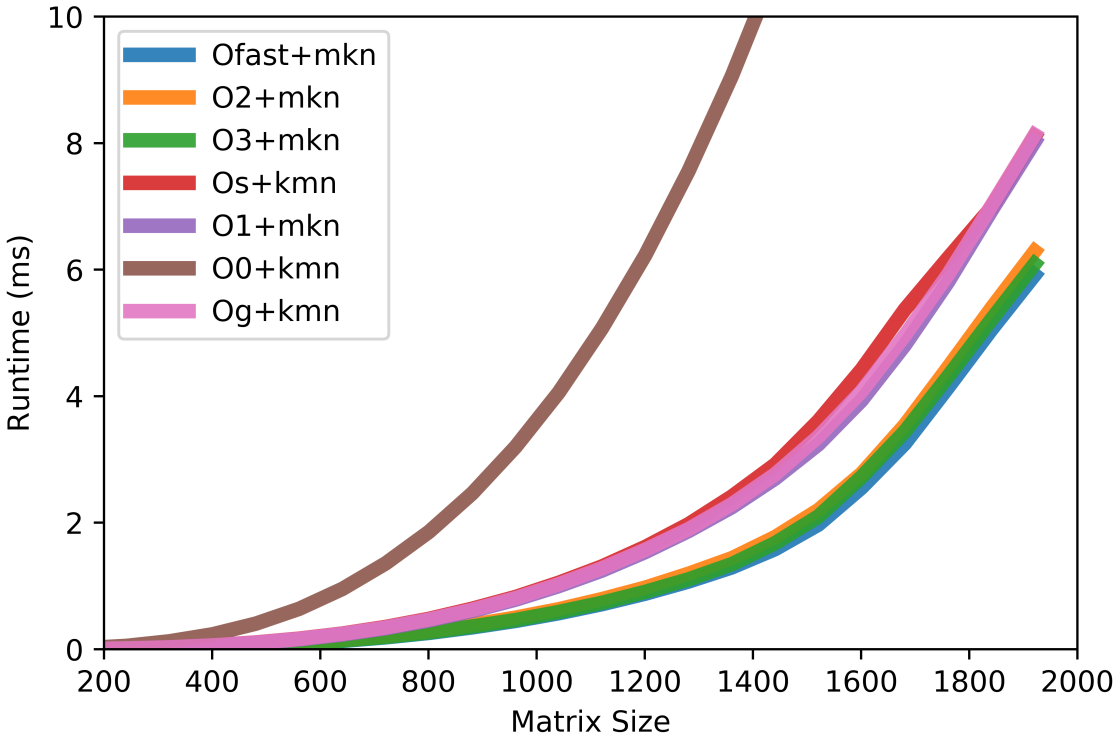

Figure 4: Best permutations for each optimization

3.2 Analyzer

In order to check if the results in the previous sections and if our expectations regarding the loop order are correct, we perform experiments using the sun studio analyzer. We run it with the following values: `m` = `n` = `k` = 1300. The results obtained are the following:

Permutation	L1-Hit	L1-Misses	L1-miss/Hits	L2-Hit	L2-Misses	L2-miss/Hits
kmn	49659752577	12804006	0.000257834	6402003	7007008	1.09450
knm	45375460261	4401376378	0.096999	2740056857	1661661662	0.606433
mkn	49679772598	12804006	0.00025773	6402002	5005006	0.781787
mnk	47497586348	2199087689	0.0462989	1917399599	280280282	0.146177
nkm	45375460262	4401376378	0.096999	2573604804	1828828830	0.71060
nmk	47490586214	2189484686	0.0461035	2176680680	12012013	0.0055185

Table 1: Hardware counter without compiler optimization for the different loops.

Permutation	L1-Hit	L1-Misses	L1-miss/Hits	L2-Hit	L2-Misses	L2-miss/Hits
kmn	6496508599	12804006	0.0019709	6402002	6006008	0.938145
knm	8018032975	4378969371	0.54614	2660031833	1717717718	0.645750
mkn	6656669057	12804006	0.00192348	6402003	4004006	0.625430
mnk	5015024339	2551211205	0.508713	2269509714	289289292	0.127467
nkm	7977992849	4388572375	0.550084	2704845876	1682682683	0.622099
nmk	4734743542	2631237888	0.555729	2618433809	22022024	0.00841038

Table 2: Hardware counter with compiler optimization for the different loops.

We can observe in Tab.1 and in Tab. 2 that the Analyzer tools are confirming our assumptions. The lower the ratio misses/hits in the L1 cache, the better the performance. That is, for the `mkn` and `kmn` we obtain the lowest ratios, therefore the highest performance.

Our results indicate something strange though, the L1-miss/Hit ratio for the optimized version was lower than the unoptimized one. That's because the cache ratio doesn't tell us the full story, so we had to check more hardware counters. To achieve that, we used the 'perf' tool and took a sample 5 times for each function. And the results are the following.

Permutation	Cache ref	Cache misses	Instruction	Branches	Branch misses	Time elapsed
kmn	35.455 M/s	1.318 %	3.76 ins/cycle	266.989 M/s	0.08 %	8.354626
knm	24.178 M/s	0.673 %	2.64 ins/cycle	188.468 M/s	0.08 %	11.827570
mkn	34.661 M/s	1.332 %	3.74 ins/cycle	267.771 M/s	0.08 %	8.330332
mnk	23.481 M/s	1.276 %	2.56 ins/cycle	182.687 M/s	0.08 %	12.201672
nkm	24.146 M/s	1.613 %	2.59 ins/cycle	185.760 M/s	0.08 %	12.002495
nmk	29.362 M/s	2.507 %	3.13 ins/cycle	222.541 M/s	0.08 %	10.020521

Table 3: Hardware counters without compiler optimization for the different loops.

Permutation	Cache ref	Cache misses	Instruction	Branches	Branch misses	Time elapsed
kmn	225.672 M/s	1.069 %	3.60 ins/cycle	875.275 M/s	0.16 %	3.843357
knm	74.432 M/s	0.834 %	1.87 ins/cycle	578.108 M/s	0.08 %	3.862169
mkn	225.013 M/s	0.649 %	3.67 ins/cycle	892.828 M/s	0.16 %	3.769229
mnk	98.272 M/s	0.695 %	1.88 ins/cycle	775.164 M/s	0.08 %	5.733300
nkm	73.003 M/s	1.399 %	1.82 ins/cycle	561.706 M/s	0.08 %	3.975076
nmk	107.817 M/s	0.959 %	2.02 ins/cycle	835.584 M/s	0.08 %	5.3215119

Table 4: Hardware counters with compiler optimization for the different loops.

Tab.3 and Tab.4 also confirm our initial assumptions, showing higher cache references and branches for the permutation `kmn` and the `mkn`. Comparing Tab.3 and Tab.4 we can clearly identify that the optimized version, Tab.4, has a huge improvement on cache references and branches across all the permutations. This explains the much shorter execution time of our optimized version. Having a higher cache reference means that the application is using the caches more while having a higher branch count means that the CPU can utilize its branch prediction features that it has so it can do speculative execution and speed up the program.

4 Blocking

The blocked version of the matrix multiplication function, `matmult_blk()` takes block size as argument. This allows testing different values and find the optimal block size. The previous matrix multiplication version is producing $3m^2$ data values. The problem is that an entire matrix might not fit into a small memory. Therefore the idea is to split the work into small chunks of computation. When using the following code we referred to $3bs^2$ data values. The main asset of this version is that we can choose bs small enough so that it fits into the small memory. The bloc size (bs) that we use is decided depending on the cache size. The first two things we should decide to implement the blocking are the loop order, and the block size. The version that we will use for the loop order is the one that gave us the best performance in the previous section, that is, the `mkn`. Here we present the code that we implemented for the blocked version of the matrix multiplication function:

```
void matmult_blk(int m, int n, int k, double **A, double **B, double **C, int bs)
{
    int i, j, p, i0, j0, p0, min_i, min_j, min_p;
    bs = (bs <= 0) ? 1 : bs;
    for (i = 0; i < m*n; i++){
        C[0][i] = 0.0;
    }

    for (i = 0; i < m; i += bs){
        for (j = 0; j < n; j += bs){
            for (p = 0; p < k; p += bs){
                min_i = MIN_(m, i+bs);
                min_j = MIN_(n, j+bs);
                min_p = MIN_(k, p+bs);
                for (i0 = i; i0 < min_i; i0++){
                    for (p0 = p; p0 < min_p; p0++){
                        for (j0 = j; j0 < min_j; j0++){
                            C[i0][j0] += A[i0][p0] * B[p0][j0];
```

```
}   }   }   }   }   }   }
```

Our aim is to optimize the block size with respect to the L2 cache size (1024 KB in our case). We do it around the L2 because as we have seen, there is not a significant drop in performance regarding the L1 cache. To select the block size we impose that the memory used in the block is closed to the cache but a bit less to leave some free space to auxiliary values. We should compute how many `doubles` are used in each block. We see that each indexing (`i`, `j`, `p`) only takes the `bs` values in each bloc, that is $\texttt{bs}^2$ numbers are used from each matrix. As we have three matrices, $3 \cdot \texttt{bs}^2$ `double` values are used. With this number and knowing that the L2 cache size is 1024 KB, we estimate an optimal block size `bs` of 206 since is the largest option that is not higher than the cache size, but it also leaves memory for other variables. In the following figure we present the performance for huge matrices depending on the block size.

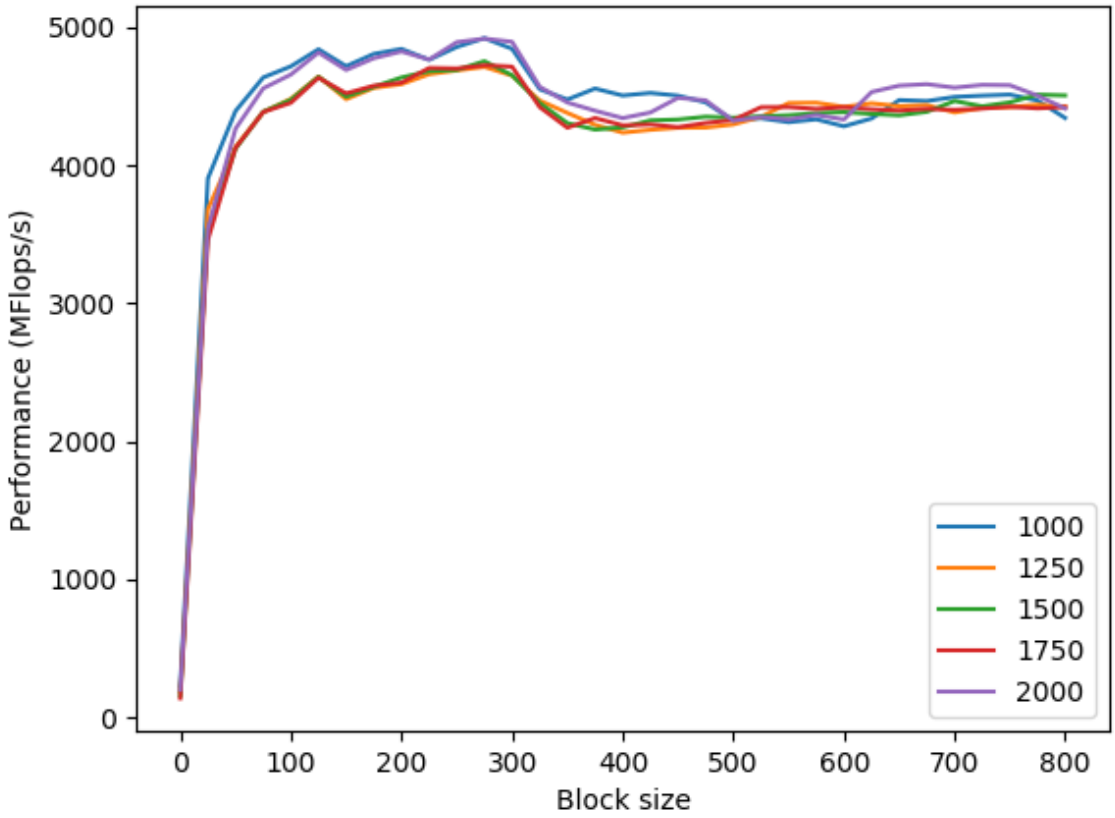

Figure 5: Performance for different block sizes.

In Fig.5 we observe that the highest performance is for a block size of about 200-300. We can see that for 300 is slightly better. Although the optimal one is around 200, this does not explicitly mean that other block sizes can give better performance, as it happens in our case. This can be due to prefetching and other actions in the chips.

When the size of the matrices is smaller than any of the cache sizes, blocking makes no difference. Nevertheless, is not until the size goes beyond the L3 cache that the blocking actually improves the performance. This is shown in the following figure:

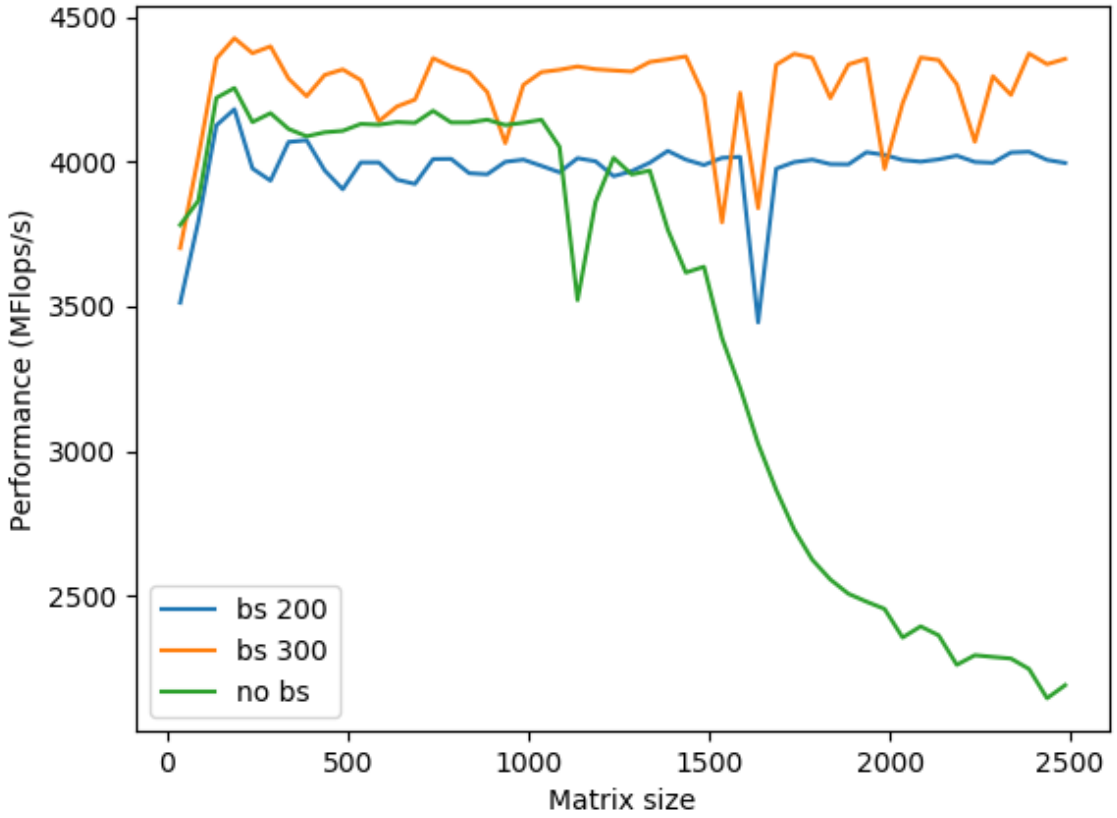

Figure 6: Performance for different block sizes.

We can observe in Fig.6 that with matrices smaller than the cache size, blocking is not worth it whereas when we go beyond the cache size, the performance using blocking remains steady, while the performance without blocking drops. We have computed the performance for two different block sizes. The `bs` $= 200$ corresponds to the theoretical value and the `bs` $= 300$ is the one that gave the best performance in the experiments. We observe that for the theoretical block size we obtain the expected performance. That is, for small matrices, without blocking the performance is better, but once we achieve the cache limit, we get a better performance with blocking.

5 Conclusions

We have tested the performance of the matrix-multiplication using different implementations. First of all we did the matrix multiplication in the normal way, and we initialized the variables in the inner loop. Using this implementation we obtained the best performance. Afterwards, we have tried different permutations for the loops, reaching the conclusion that the permutations with `n` as the most inner loop are the ones that gave the best performance.

Second of all we have proved that the performance drops when the size of the data increases. Nevertheless, using the best permutation we can compensate the dropping in the performance due to the matrix size. Once we pass through the second cache level, the best permutation has still a good performance. We also proved that the best permutation has a good performance both with and without optimization. In order to

check these results we used an analyzer tool and studied the ratio misses/hits. We obtained that the best permutation has the lowest ratio.

The last thing we tried to improve the performance was the blocking method. As we expected, this method shows and improvement in the performance when the matrix size exceeds the cache size. When the matrix size is small, we obtain a worse performance using blocking.

To sum up, we have implemented several tuning and optimization methods to study the performance of the matrix multiplication. Nonetheless, the performance using the CBLAS library was the best one with any doubt.

Appendix

Cluster specifications

- `Architecture: x86_64`
- `CPU op-mode(s): 32-bit, 64-bit`
- `Byte Order: Little Endian`
- `CPU(s): 24`
- `On-line CPU(s) list: 0-23`
- `Thread(s) per core: 1`
- `Core(s) per socket: 12`
- `Socket(s): 2`
- `NUMA node(s): 2`
- `Vendor ID: GenuineIntel`
- `CPU family: 6`
- `Model: 85`
- `Model name: Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz`
- `Stepping: 4`
- `CPU MHz: 2600.000`
- `BogoMIPS: 5205.08`
- `Virtualization: VT-x`
- `L1d cache: 32K`
- `L1i cache: 32K`
- `L2 cache: 1024K`
- `L3 cache: 19712K`
- `NUMA node0 CPU(s): 0-11`
- `NUMA node1 CPU(s): 12-23`